Scaling Data Race Detection for Partitioned Global Address Space Programs

Abstract

Contemporary and future programming languages for HPC promote hybrid parallelism and shared memory abstractions using a global address space. In this programming style, data races occur easily and are notoriously hard to find. Previous work on data race detection for shared memory programs reports 10X-100X slowdowns for non-scientific programs. Previous work on distributed memory programs instruments only communication operations. In this paper we present the first complete implementation of data race detection at scale for UPC programs. Our implementation tracks local and global memory references in the program and it uses two techniques to reduce the overhead: 1) hierarchical function and instruction level sampling; and 2) exploiting the runtime persistence of aliasing and locality specific to Partitioned Global Address Space applications. The results indicate that both techniques are required in practice: well optimized instruction sampling introduces overheads as high as 6500% (65X slowdown), while each technique in separation is able to reduce it to 1000% (10X slowdown). When applying the optimizations in conjunction our tool finds all previously known data races in our benchmark programs with at most 50% overhead. Furthermore, while previous results illustrate the benefits of function level sampling, our experiences show that this technique does not work for scientific programs: instruction sampling or a hybrid approach is required.

1. Introduction

Attaining good performance and efficacy on contemporary and future large scale High Performance Computing systems requires combining multiple levels of parallelism: *intra-node* parallelism is usually exploited using shared memory programming models, while *inter-node* parallelism is exploited using message passing or shared memory abstractions. This is illustrated by the large body of research in hybrid programming models: OpenMP+MPI, UPC+MPI, Intel TBB + MPI and OpenMP+UPC.

In this parallelism rich environment, bugs due to non-deterministic execution and conflicting memory accesses are fairly common and notoriously hard to detect. Previous work demonstrates the ability of dynamic program analyses to find concurrency bugs (data race [39], atomicity violations [27], deadlock [7]) in shared memory programs. Dynamic program analyses have been also used to find *heisenbugs* in distributed memory programs: DAMPI [44] for MPI wildcard receives and UPC-Thrille [35] for data races in Unified Parallel C [10].

Data race detectors for shared memory programming [37, 39] track individual memory references and implement a centralized analysis to reason about program semantics: their implementations are heavily optimized to reduce the instrumentation overhead and are able to function with overhead lower than 10X. Bug finding [35, 44] for distributed memory programming models is made scalable by using a distributed analysis, but the current approaches illustrated by DAMPI and UPC-Thrille track only the calls into communication libraries. Thus, distributed memory tools provide only limited functionality and need to be extended with tracking of memory references. Furthermore, while acceptable when testing programs on workstations, the current overhead of dynamic program analyses is hard to stomach at the contemporary HPC concurrencies of tens of thousands of cores. Large scale analyses face the additional challenge to provide the lowest achievable overhead while still providing good coverage. While the adoption criteria for shared memory tools is "acceptable overhead", more stringent optimality criteria are desired at scale.

In this paper we present and analyze the first complete dynamic analysis for distributed memory programs able to track both memory references and communication calls. We extend the UPC-Thrille data race detection tool with tracking of individual memory references and discuss the techniques required to achieve low overhead for scientific applications running at scale. The results are validated for implementations of the NAS Parallel Benchmarks [6] as well as other fine-grained dynamic programming and tree search applications. Although validated only for UPC benchmarks, we believe that our findings are widely applicable to any tool for data race detection in Partitioned Global Address Space languages: Chapel, Titanium, Co-Array Fortran, X10.

UPC-Thrille, detailed in Section 2.2, implements a dynamic program analysis running in two phases. In the first phase the program is executed with additional instrumentation and data about memory accesses, communication and task synchronization operations is gathered and analyzed. For the purposes of this paper we distinguish three types of overhead: 1) *instrumentation overhead* is introduced by the checks to prune the non-interesting data accesses; 2) *computation overhead* is introduced by the operations on internal data structures to manage the interesting accesses; and 3) *communication overhead* introduced by the exchange of conflicting accesses between tasks.

The most widely used technique to reduce overhead is sampling [3, 4, 19, 28, 35, 44] of the program execution. Tools for shared memory use instruction level sampling while the distributed memory tools [35, 44] implement its equivalent by sampling the communication calls. For shared memory, Marino et al [28] recently introduced LiteRace which coarsens the granularity of the sampling at function boundaries: functions are compiled in two versions, un- and instrumented, each version being selected at runtime using heuristics. LiteRace showed better scalability and coverage than instruction level sampling when applied on several Microsoft programs, as well as Apache and Firefox.

We have experimented with both instruction level sampling and function level sampling. The results presented in Section 6 indicate that instruction level sampling performs better than function level sampling for scientific programs. Instruction level sampling adds runtime overhead as high as 65X while many runs using function level sampling did not terminate, even when instrumenting only the first execution of a function. This result contradicts the trends reported for LiteRace and it is caused by a combination of two factors: 1) determining the locality of a reference is expensive in PGAS programs; and 2) scientific programs have long running loops, with billions of memory accesses per invocation in our benchmarks. The results also indicate that the instrumentation overhead dominates the computation and communication overhead of the analysis.

We propose a combination of techniques to minimize the tool overhead. In Section 4.1 we describe how to use program semantic information such as aliasing to further reduce the instrumentation overhead. In Section 4.2 we propose a hierarchical sampling approach where instrumentation is dynamically controlled both at the function level and at the instruction level. Hierarchical sampling reduces all three type of overhead: instrumentation, computation and communication. Applying these techniques in isolation is able to reduce the maximum tool overhead from 6500% with instruction sampling to roughly 1000%. Using them in combination, we were able to reduce the maximum overhead from 1000% to only 50% while finding the same races.

2. Background

2.1 Unified Parallel C

UPC is a parallel extension of the ISO C programming language for high performance computing. UPC uses the Single-Program-Multiple-Data (SPMD) programming model and provides a Partitioned Global Address Space: each task has access to a private address space and to a global shared address space. The language extends the C type system with the qualifier of `pointer-to-shared` to denote accesses to the global address space. `Pointers-to-shared` can be casted to proper C pointers, but not vice-versa. This is widely used in practice for performance reasons and for calling into libraries such as Intel MKL. In addition, the language provides synchronization primitives (`lock, barrier`), bulk memory transfers (`memput, memget`), as well as a memory consistency [24, 48] model.

Together with Chapel, X10, Co-Array Fortran and Titanium, UPC belongs to the family of Partitioned Global Address Space languages. These PGAS languages distinguish between local and global references and provide support for logical data layouts, such as block-cyclic array distribution. As a result, they implement complicated memory management and a reference to a global object is orders of magnitude slower than a load/store instruction due to complex addressing rules.

2.2 Finding Data Races with UPC-Thrille

A *data race* occurs in parallel programs when two threads access the same memory location with no ordering constraints between them, and at least one of the accesses is a write [1, 33].

Before this work, UPC-Thrille was able to find only data races between accesses performed using variables with the `pointer-to-shared` data type (e.g. `shared int *`) and communication calls (e.g. `memget`). It implements an active testing [21] methodology which works in two phases:

- A predictive analysis phase which uses a distributed lockset-based algorithm [35, 39] to identify potential data races. As only references with a `pointer-to-shared` data type are tracked, the implementation amounts to instrumenting calls into a communication runtime. The results reported show that UPC-Thrille is able to find bugs with good scalability and overhead lower than 15%.
- A confirmation phase, where the program is re-executed under a controlled schedule that attempts to make the potential races manifest. As the first phase reports a small number of candidates, this phase is often skipped in practice.

Well optimized UPC programs usually cast `pointers-to-shared` (e.g. `shared int *`) to C proper pointers (e.g. `int*`) and the released UPC-Thrille misses a large class of data races introduced by memory aliases. Furthermore, the presence on non-blocking communication operations [8] introduces another class of data races. As non-blocking communication is a "background" asynchronous activity that can be overlapped with computation, memory accesses within a task can race with the communication operations initiated by the same task. MPI programs face a similar problem when using the MPI_Isend/IRecv non-blocking communication primitives. A complete solution for finding both "traditional" and races introduced by non-blocking communication needs to track all the memory references, including those using C pointers, as well as communication calls.

3. The Overhead of Data Race Detection

Runtime overhead due to instrumentation is recognized as a problem that dynamic race detectors have to address. Commercial tools for C programs such as the Intel Thread Checker or the Sun Thread Analyzer, usually provide full coverage at the expense of 600X execution slowdown [41] on scientific OpenMP programs with small memory footprints. Average overheads on other scientific programs for the Intel Thread Checker have been reported [38] around 200X and as high as 485X.

Sampling techniques have been introduced by Arnold and Ryder [3] and later adopted in other bug finding tools [4, 28] for parallel programs. The efficacy of these techniques is determined by the granularity of the instrumented code region and the sampling strategy. Tools [4] for finding bugs in programs running on managed runtimes (e.g. Java) tend to use instruction level sampling; the additional instrumentation overhead is not perceived as unacceptable since the runtime already manages object metadata and access. These systems usually observe up to 3X slowdowns for non-scientific applications and data is not available for HPC applications. The equivalent of instruction sampling is performed in distributed memory tools such as DAMPI [44] and UPC-Thrille [35] which track communication calls.

Recently, Marino et al [28] proposed a technique to coarsen the sampling control from instruction level to function level. They use a compiler to generate instrumented and un-instrumented versions of functions and select the appropriate copy at runtime. The instrumented version of a function monitors every memory reference during its execution. Their LiteRace tool introduces up to 3X overhead while providing good coverage on non-scientific programs; it has not been evaluated on scientific programs. In the rest of this paper we refer to this technique as function sampling. One reason that function sampling outperforms instruction sampling is that it amortizes better the cost of tracking memory references: function sampling executes one branch/decision per function call while instruction sampling executes one branch/decision per instruction.

Several sampling strategies have been proposed and evaluated for non-scientific programs. Random sampling has been shown to provide poor coverage. SWAT [19] detects memory leaks and uses an approach where the execution of code segments is sampled at a rate inversely proportional to their execution frequency. LiteRace uses a bursty sampler, where the execution of a function is sampled initially at a 100% rate and the sample rate is progressively reduced until it reaches a lower bound. Both approaches try to give priority to regions of code rarely executed and give priority to the first execution of any code region.

The implementation of UPC-Thrille described in [35] uses "instruction" level instrumentation with a bursty sampling similar to LiteRace. UPC-Thrille instruments every memory access performed using `pointer-to-shared`, either at word granularity or using bulk `memget/memput` memory operations.

In order to provide a complete data race detection solution we have modified UPC-Thrille and the Berkeley UPC compiler to track all memory references, including all references through C proper pointers. We provide a well optimized implementation of instruction sampling that makes extensive use of C macro-definitions to eliminate function call overheads for the instrumentation code. Every memory reference is examined using a bursty sampling strategy. We have also implemented function sampling with the same bursty strategy.

For any sampled memory reference, the implementation checks whether the address resides within a thread's private address space or within the global address space. This check requires integration with the UPC runtime memory management module and it is an expensive operation, common to PGAS languages. References to the private address space are ignored as they cannot race. Global references are inserted into the UPC-Thrille internal data structures and further checked against other references.

We distinguish three types of overhead: 1) *instrumentation overhead* is introduced by the checks to prune the non-interesting data accesses; 2) *computation overhead* is introduced by the operations on internal data structures to manage the interesting accesses; and 3) *communication overhead* introduced by the exchange of conflicting accesses between tasks. Thus, private references contribute only to instrumentation overhead while global references also contribute to the computation and communication overhead.

We have performed experiments with instruction sampling using the default settings reported in [35] and with multiple function sampling strategies. For brevity we do not present detailed results. Our results indicate that for the NAS Parallel Benchmarks function sampling is not a scalable strategy. For most benchmarks class B and up, experiments when instrumenting only the first invocation of a function did not terminate: some exhausted the available memory while some were manually terminated after observing 1000X slowdown. Results for instruction level sampling indicate that this approach is able to find races with up to 65X slowdown. Detailed results are presented in Section 6.

This behavior contradicts the intuition that function sampling scales better than instruction sampling. The performance reversal is caused by the too coarse granularity of control over instrumentation: as loops within these benchmarks execute billions of references, function sampling tracks billions of references.

4. Techniques to Reduce Overhead

As function level sampling does not work and instruction sampling introduces a 65X overhead that is unacceptable when running at scale, our implementation uses two techniques to reduce the number of tracked memory references without sacrificing the precision of the analysis.

The first optimization reduces the overhead of instrumentation by exploiting the insight that *aliases are persistent* in PGAS programs: once one is created it will point in the same region (private or global) for a long period of time. Using this we can eliminate the overhead introduced by looking up the physical memory layout inside the language runtime.

The second optimization reduces overhead using hierarchical sampling. By combining function and instruction sampling we amortize the cost of instrumentation while retaining fine grained control over the number of events sampled.

4.1 Exploiting the Persistence of Locality

PGAS languages, such as UPC, Titanium, CAF, Chapel and X10, provide the abstraction of a shared memory address space. Data residing in this space is accessible through references to variables that have a particular type, e.g. `pointer-to-shared`" type in UPC or "global" in Titanium[1].

The memory management inside any PGAS language runtime is complex due to the need to provide globally addressable memory and to support data layouts, e.g. block cyclic layouts. Thus, a reference to a global object is orders of magnitude [20] more expensive than a local reference, through a C pointer in the UPC case. Application developers aggressively cast global references to local and compiler optimizations [22, 25, 26] have been explored to "privatize" global references.

For every local memory reference, the data race detection code needs to perform the inverse up-cast operation and check whether the address is globally visible. This operation is also orders of magnitude more expensive than a regular memory load/store.

In our implementation we limit the number of up-casts performed at runtime using the intuition that aliases/locality are persistent: during the program execution a reference will access only the private space or only the global space, independent of its static data type. This assumption allows the analysis to determine at runtime the "locality" of any reference only once and cache the result for the rest of the execution. In our implementation, we add a shadow variable to cache the locality of every memory reference expression.

The persistence of locality assumption is valid in all of our test programs and it does not decrease the precision of the analysis. The heuristic may lead to false negatives (miss real data races) when the underlying assumption is not valid for the program. However, the technique can be trivially generalized for programs with a more dynamic behavior. As casts in PGAS languages are complicated and are implemented as runtime "calls", any casting call can be modified to invalidate the locality information cached. The performance of this approach is determined by the ratio of casts to memory references performed by the program at runtime. The additional overhead for realistic programs is likely to be negligible in practice.

4.2 Hierarchical Sampling

For every memory reference there are two sources of runtime overhead. Instrumentation overhead is introduced to decide whether the reference should be recorded and computation overhead is introduced when recording the reference in the tool internal data structures. By reducing the instruction sampling rate one can clearly reduce overhead, but at the expense of program coverage. To provide both low overhead and good coverage we propose a hierarchical sampling approach which combines the fine grained control of instruction sampling with the overhead amortization provided by function sampling. By using a good hierarchical sampling strategy, we can reduce the instrumentation overhead while retaining the ability to sample from a diverse context with less redundancy. Using the concept of code regions, we formally define instrumentation and hierarchical sampling.

Definition 1 (Code regions). *We inductively define code regions. By definition, the smallest unit of a code region is a memory reference (read or write). A code region is a reference or a sequence of one or more code regions. The entire program is the largest code region. Each code region R has a label, denoted as $\#R$.*

Functions, loop bodies, basic blocks etc. are examples of code regions. We assume structured code, i.e. that all code regions are properly nested.

Definition 2 (Region stack). *During program execution, a region stack RS is maintained. Similar to a call stack, when a region is entered, the label of the region $\#R$ is pushed to RS. When exiting a region, the last label is popped from the stack. At the beginning of a program execution, RS is initially empty.*

Definition 3 (Instrumentation). *Instrumentation is a transformation of a code region $R \rightarrow R^{inst}$.*

If R is a memory reference (base case)

```
$R^{inst}$ =    if check-reference($\#R :: RS$) then
                    log(#R)
                R
```

[1] Actually, in Titanium any reference is global by default and the language provides local qualifiers.

Else, if R is a sequence of regions $[R_1, R_2, \ldots, R_n]$,

```
             if check-region($\#R :: RS$) then
                 $RS = \#R :: RS$;
$R^{inst}$ =         $[R_1^{inst}, R_2^{inst}, \ldots, R_n^{inst}]$;
                 $RS = tail(RS)$
             else
                 $[R_1, R_2, \ldots, R_n]$
```

By specializing the *check-reference* and *check-region* and choosing the region granularity, we can implement multiple sampling algorithms. For example, instruction sampling with an exponential backoff (strategy **I** in the experiments presented in Section 6), is implemented as the following functions. The map $p : label \rightarrow R$ contains the (dynamic) sampling probabilities of regions.

```
$\forall \#R \in Statements.\ p(\#R) = 1.0$

check-reference($\#R :: RS$) = if rand() < $p(\#R)$ then
    $p(\#R) *= BACKOFF\_FACTOR$;
    return $true$
else
    return $false$

check-region(x) = true
```

Function sampling as introduced by the LiteRace [28] implementation is defined as follows. The region is a whole function and the *sample-strategy* function depends on the strategy of sampling, such as a fixed probability, random or an adaptive strategy.

```
check-reference(x) = true

check-region$\#R :: RS$) = sample-strategy($\#R$)
```

Intuitively, the *check-reference* function decides what events should be logged at runtime, while the *check-region* function provides control over the granularity of these decisions. We propose a hierarchical sampling strategy that combines instruction sampling with function sampling. The combination of hierarchical sampling with the aliasing runtime heuristic is referred to as **HA** and described as:

```
$\forall \#R \in Statements \cup Functions.\ p(\#R) = 1.0$

check-reference($\#R :: RS$) = if $p > 0 \wedge$ rand() < $p(\#R)$
then
    if is-local-access(R) then
        $p(\#R) = 0$ ;                    // alias heuristic
        return $false$;
    else
        $p(\#R) *= STMT\_BACKOFF\_FACTOR$;
        return $true$;
else
    return $false$

check-region($\#R :: RS$) = if $p > 0 \wedge$ rand() < $p(\#R)$ then
    $p(\#R) *= FUNC\_BACKOFF\_FACTOR$;
    return $true$
else
    return $false$
```

This implementation uses exponential backoff at both individual reference and function granularity.

5. Benchmarks

We evaluate UPC benchmarks using fine-grained and bulk communication. For implementations using bulk communication primitives we use the NAS Parallel Benchmarks (NPB) [6, 31, 32], releases 2.3, 2.4, and 3.3. We have performed experiments with the problem classes A, B and C and D; overall the memory footprint of the workload varies from tens of MBs to tens of GBs. Asanović et al [5] examined six different promising domains for commercial parallel applications and report that a surprisingly large fraction of them use methods encountered in the scientific domain. In particular, all methods used in the NAS benchmarks (multigrid, sparse-matrix operations, sorting, Fast Fourier Transformation, dense linear algebra) appear in at least one commercial domain. Thus, beside their HPC relevance, these benchmarks are of interest to other communities.

The fine-grained benchmarks reflect the type of communication/synchronization that is present in larger applications during data structure initializations, dynamic load balancing, or remote event signaling. The *guppie* benchmark performs random read/modify/write accesses to a large distributed array, a common operation in parallel hash table construction, graph algorithms and data mining. The amount of work is static and evenly distributed among tasks at execution time. The *psearch* benchmark performs parallel unbalanced tree search [34]. The benchmark is designed to be used as an evaluation tool for dynamic load balancing strategies.

The selected programs provide a good sample of various programming and software engineering styles, dynamic application behavior and scalability characteristics. The NAS benchmarks contain many function calls and have a structure common to any large application. The fine-grained benchmarks contain few or no user defined function calls, a structure common to many scientific libraries and their unit testing. In the NAS benchmarks, the ratio of local memory accesses to communication calls performed at runtime is large ($O(10^5)$ or above) , while in the fine grained benchmarks they are roughly equal. The NAS benchmarks implement iterative methods, while the code in the fine-grained benchmarks represents a "direct" solve executed only once. NAS FT exhibits a race between the initialization code and the subsequent computation: initialization is executed only once. *guppie* performs random updates to a global table and it exhibits two races: read-write and write-write. *psearch* implements a work stealing strategy that exhibits random data races. Due to the randomness of the access pattern, these benchmarks require the tool to provide good program coverage.

The same workload has been evaluated in [35] where the authors describe a larger set of races than that used in this section for illustrative purposes. Our extended implementation finds all these and, in addition, uncovers several other races. For a summary please see Table 1. For example, we detect a previously unknown race in NAS CG introduced by the presence of aliasing: memory is initialized using "local" pointers and distributed without synchronization to other tasks using global pointers. In NAS BT and SP we uncover seven and nine additional races, respectively. These races are real and confirmed by the tool; they occur when executing custom synchronization code:
`signal (v = 1);` $\|$`wait(while(v == 0); );`.

The 44 new data races found in NAS LU are caused partially by custom synchronization code and partially by data references separated by custom synchronization code. We are still classifying these races and we will provide a detailed breakdown in the final version of the paper. Note that identifying races in the presence of custom synchronization code is a common limitation of data race detection tools.

6. Evaluation

Our implementation extends UPC-Thrille as contained in the Berkeley UPC release 2.14.2 with load/store tracking, hierarchical sampling and the runtime alias disambiguation heuristic.

The experimental results are obtained on a Cray XE6 system composed of nodes containing two twelve core AMD MagnyCours 2.1 GHz processors. The system has two nodes attached to a Gemini network interface card, forming an overall 3-D torus network with 6,384 nodes. The network is providing a bandwidth of 9.375 GBytes/sec per direction in 10 directions. The maximum injection bandwidth per node is 20GB/s.

We evaluate the performance of our data race detection tool on 10 UPC programs written in different programming styles. For each benchmark we evaluate the overhead of several configurations of the tool. Instruction sampling is denoted by **I** and for this configuration we report results with the default setting of 0.9 instruction backoff factor, reported in [35]. Using this setting, the authors report overheads lower than 15% when running at scale. Function sampling is denoted by **F**, while hierarchical function and instruction sampling is denoted by **H**. For hierarchical sampling, instructions are sampled with the default values for **I**, while the numbers in the title denote the function backoff factor. Thus, **H1** is identical to **I** (always samples functions), while with **H0** we sample only the first invocation of any function. At the mid-point **H.5** the probability of sampling a function invocation decays from 1 by 0.5 each time the function is sampled; for long running programs the sampling probability converges to 0. The letter **A** in the configuration name denotes applying the aliasing heuristic to that particular sampling method.

6.1 Comparison of Sampling Techniques

We illustrate the differences between the different tool configurations using the CG benchmark. These trends are representative for the whole suite of benchmarks we examined. For reference, the original UPC-Thrille tool adds 8% runtime overhead when instrumenting only communication calls (labeled as **NL** in the graphs for No-Local). Our implementation finds one additional race in the implementation of this benchmark when compared to the original UPC-Thrille.

Figure 1 presents the tool performance when applied to the CG benchmark classes A and D running on 16 and 2048 cores respectively. The benchmark implements an iterative method and Class A solves a problem with a small memory footprint (MBs) in few iterations, while class D solves a large (GBs) problem. Previous other shared memory data race detectors [16, 28, 37, 39] have been scaled at most up to 16 cores and on applications using small data sets. LiteRace is validated on a four core system, while the tool presented by Raman et al [37] has been scaled up to 16 cores.

Instruction level sampling **I** of all memory references adds a 3700% overhead to the CG benchmark execution. This is obtained using the default sampling frequency recommended by the tool authors to find races when instrumenting only communication calls. The overhead can be reduced by decreasing the sampling frequency, albeit at the expense of code coverage.

Function level sampling **F.5** introduces a 3000% overhead for class A, lower than the 3700% overhead of **I**. A comparison of the overhead breakdown for **F** and **I** illustrates the fundamental differences between the two methods. **I** introduces 3600% instrumentation overhead, while **F.5** adds only 100% instrumentation overhead. This large difference validates the common intuition that function level sampling amortizes better the cost of deciding what references to track. On the other hand, **F.5** exhibits a large 2900% computation overhead to record and reason about the memory references that are actually tracked. The computation overhead for **I** is very small at 2%. This behavior is explained by the temporal distribution of tracked memory accesses during the program execution. UPC-Thrille uses a combination of lockset based and happens-before analysis that requires tracking all memory references between two `barrier` statements. Function level sampling exhibits a clustered behavior, where many memory references are tracked for a short period of time. Instruction sampling spreads the tracking of memory references more evenly over the program execution. Thus, the behavior of function sampling is determined by the scalability of the tool internal data structures, while the behavior of instruction sampling is determined the speed of "classifying" a memory access. We discuss the scalability of data structures in Section 6.2.

Hierarchical sampling **H.5** provides better performance than both function and instruction sampling and exhibits 2650% overhead. Most of this overhead is instrumentation overhead.

Adding the aliasing heuristics to any of the tool methods greatly improves performance. The overhead of instruction sampling is reduced from 3700% to 205% with **IA**. The overhead of hierarchical sampling is reduced from 2650% with **H.5** to 199% with **HA.5** and from 394% with **H0** to 17% with **HA0**. The lowest overhead of data race detection for the CG class A benchmark running on 16 cores is obtained by the **HA** approach.

Similar trends are observable when scaling the problem and running class D on 2048 cores. For this particular configuration, the **F** and **FA** methods do not terminate due to out of memory errors or excessive slowdown. **I** exhibits a 359% slowdown, while all hybrid methods **IA** and **HA** exhibit less than 15% slowdown.

6.2 Implementation Overheads

Previous work on data race detection focuses on word-level memory accesses and require only keeping track of conflicting addresses. These tools usually use internally hash table data structures. For scientific programs with bulk communication operations (PGAS or MPI), data races on full memory ranges can occur during execution. UPC-Thrille uses an efficient Interval Skiplist [18] data structure to represent memory ranges and the authors demonstrate good performance when sampling communication operations.

As the performance of function sampling is clearly hampered by the internal data structure overhead, we evaluate the scalability using micro-benchmarks for the insertion and search operations. The time complexity of these algorithms is dependent on the number of elements in the data structure and the distribution of the intervals. We evaluate performance across a range of list sizes and interval distributions: sequential, reverse sequential, strided and uniform random. Sequential streams are often encountered in code that performs data structure initialization, and are present in all of our benchmarks. They are also the holy-grail of cache optimizations. Strided accesses occur in the Fast Fourier Transform code NAS FT, while random accesses of the form `a[b[i]]` appear in sparse methods NAS CG and sorting NAS IS, as well as *guppie*. For a real-world perspective, we also measure the average number of memory intervals that are recorded in our benchmarks.

Figures 2 and 3 present the measured performance on one core of the Cray XE6 system. For a uniform random distribution of 20,000 ranges, the average insert time is 12 μs and the average search time is 1.3 μs. For a more regular distribution of ranges such as a sequential one (e.g. [0, 10), [10, 20), [20, 30), . . .), the insertion and search times were higher at 114μs and 2.4μs, respectively. This is a weakness of the Interval Skiplist which relies on randomness of data for balancing link levels. The effect can be offset by adding some irregularity, such as inserting a mix of two different sequential streams. In the application benchmark, the memory access stream does have irregularity, and as illustrated by the results for MG inserts are on average 45μs and searches 0.54μs.

When using instruction sampling for the application benchmarks, the Interval Skiplists never grew too large. They remained

				Overhead				
Bench	LoC	Runtime(s)	#Races	NL	HA.5	IA	FA0	I
guppie	271	19.070	2 + 0	54.9%	54.2%	53.7%	DNF	74.9%
psearch	803	0.697	3 + 1	2.48%	10.8%	666%	8.01%	6490%
BT	9698	189.48	8 + 7	0.574%	1.16%	77.6%	DNF	-
CG	1654	39.573	2 + 1	1.09%	27.6%	57.6%	DNF	2579%
EP	678	54.453	0	-0.618%	0.805%	2.09%	4.74%	111%
FT	2289	62.663	2 + 0	0.601%	30.1%	121%	DNF	2744%
IS	1R36	5.130	0	0.376%	119%	159%	DNF	1201%
LU	6348	155.997	0 + 44	-0.425%	-	75.7%	DNF	-
MG	2229	18.687	2 + 4	0.336%	176%	632%	DNF	2020%
SP	5740	247.937	8 + 9	0.160%	0.861%	29.1%	DNF	-

Table 1. *Statistics for the NAS Parallel Benchmarks class C, guppie and psearch running on 16 cores. We report the races found as A + B, where A represents the number of races detected by the original UPC-Thrille tool and B represents the additional number of races detected with our extensions.*

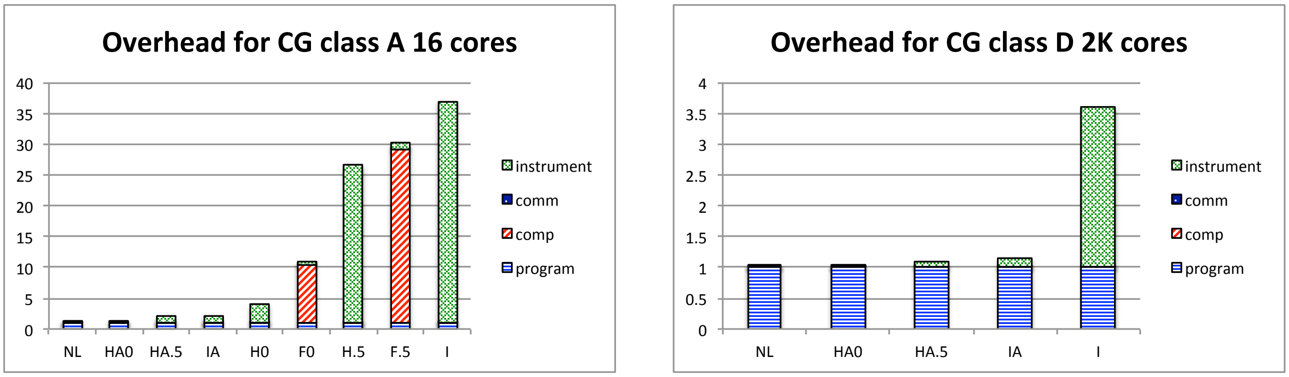


Figure 1. *Breakdown of data race detection overhead for the CG class A benchmark running on 16 cores and class D running on 2048 cores. The* **F** *and* **FA** *configurations did not finish for the class D experiment. At the mid-point* **HA.5** *the probability of sampling a function invocation decays to from 1 to 0.5, where every other invocation is instrumented.*

at under 1000 unique ranges, thus the insert and search times of the Interval Skiplist do not contribute largely to the overhead. On the other hand, when using function sampling the data structures grew above 10^6 entries, at which point we stopped the execution due to the very large overheads already accumulated.

Instruction sampling pays a higher cost for classifying a memory reference but it naturally throttles the number of references recorded at any time. Function sampling performs a fast classification while having to record a large number of references. Reference classification has a constant overhead independent of the number of references already recorded, while recording overhead scales with the number of references. This difference explains why function sampling scales worse than instruction sampling for scientific programs. For reference, when running on the Cray XE6, the average instrumentation overhead per reference is 1ns, the average memory classification is 45ns, the average computation overhead per reference is 500ns while the average communication overhead per reference is 60μs.

6.3 Scalability Aspects of Data Race Detection

The trends discussed for the CG benchmark are illustrative of the behavior of data race detection for all the other applications in our workload.

Function sampling (**F** or **FA**) is faster than instruction sampling (**I** or **IA**, respectively) for problems using small datasets, such as class A of the NAS Parallel Benchmarks. When increasing the data set size to B, C and D, function sampling in any flavor does not terminate, while the highest overhead observed for instruction sampling is 6500%. From all benchmarks considered, the only exception happens for *psearch* where **F** is roughly twice as fast than **I**. This tree search benchmark performs a constant and small amount of work per function, independent of the problem size: this is a common characteristic to many commercial applications. The performance reversal observed for most benchmarks contradicts the common intuition that function sampling performs better than instruction sampling.

Hierarchical sampling **H** performs better than both instruction sampling **I** and function sampling. While it does reduce overhead, we observe slowdowns as high as 2000% which is still unacceptable when running at scale.

Applying the aliasing heuristic reduces the overhead of data race detection for both instruction level and hierarchical sampling. The maximum slowdown observed by **IA** is 1000% while the maximum slowdown for **I** is 6500%. Similar results are observed for **HA** when compared to **H**.

Figure 4 shows the performance of our approach when performing strong scaling experiments for the classes C and D of the NAS Parallel Benchmarks. For all experiments, the lowest overhead is introduced by the **HA** configuration and we are able to find all the races with less than 50% runtime overhead when running up to 2048 cores. In the case of the NAS Parallel Benchmarks class C on 16 cores, the weighted average overhead for all the benchmarks with **HA.5** was 11.9%.

Overall, instrumentation overhead contributes the most to the slowdown caused by data race detection. The computation overhead in the scalable versions of **IA** and **HA** is small. At large scale the communication overhead is also small due to the techniques presented in [35].

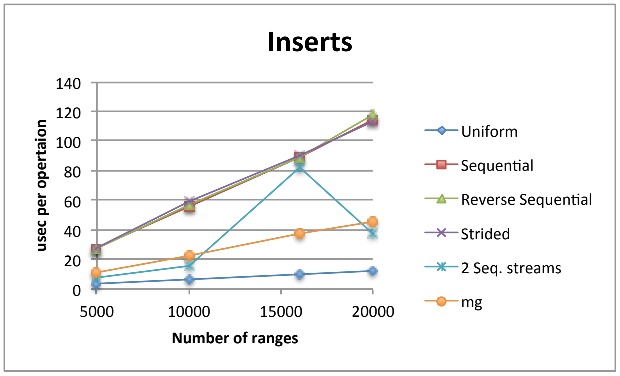


Figure 2. *Average time for the insert operation in Interval Skiplist.*

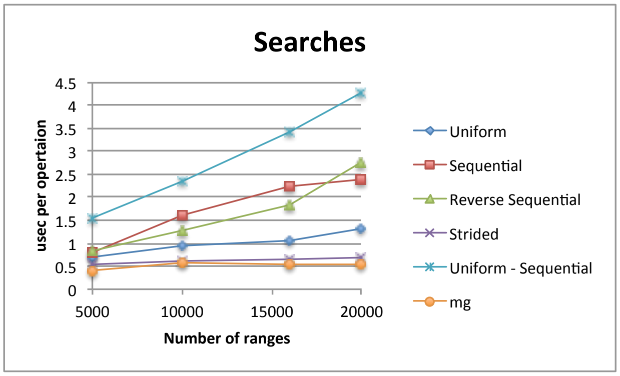


Figure 3. *Average time for the search operation in Interval Skiplist.*

7. Discussion

Sampling techniques have been shown to be effective when scaling data race detection analyses. The state-of-the-art technique applied to commercial programs is function level sampling, which our results clearly demonstrate that it is not applicable to scientific programs. Function sampling performs well when the amount of work per function is constant and scaling the data-set increases the number of function calls. Instruction or hierarchical sampling are required for scientific programs as scaling the data set increases the amount of work per function, while performing the same number of invocations. The results also indicate that a combination of hierarchical sampling and alias heuristics is required to attain acceptable overhead for data race detection on scientific programs running at scale. Any stand-alone technique does not provide a low enough overhead.

We believe that our techniques and findings are widely applicable to other languages or programming models. The alias heuristic is applicable to any PGAS language since they provide a global address space and performance and software engineering concerns require programmers to aggressively identify references that are local to a given "task". We believe that for any PGAS language the heuristic can implemented with runtime techniques since a global address contains object metadata and casting requires runtime calls. Runtime alias heuristics are not possible for MPI+OpenMP programs and they may need to be replaced by compile time analysis.

Hierarchical sampling is a generic technique orthogonal to the language, programming model employed in the application or the data race detection algorithm. It is clearly required for SPMD parallelism (UPC, CAF, MPI) or OpenMP parallel loops, where work per function scales with the problem size. When using structured parallelism as present in Habanero-C, X10 or OpenMP tasking, some applications may perform a constant amount of work per function and the overall behavior approximates commercial applications or that of our *psearch* benchmark. In this case, hierarchical sampling performs at least as well as function sampling. The benefits of hierarchical sampling are also orthogonal to the choice of data race detection algorithm: lockset based of happens-before.

There are several interesting open questions related to hierarchical and function sampling. Our hierarchical approach considers two granularities: function and instruction. In order to achieve lower overhead or to improve program coverage, one can imagine decreasing the overhead from function level to some intermediate program block level. Because of the presence of deep loop nests in scientific programs we believe that sampling as these two granularities is sufficient. A theoretical question remains whether function sampling can be made more scalable. Scalability can be improved by two approaches: i) using data structures with better scalability characteristics than Interval Skiplist; and 2) using better formalisms to reduce the number of memory references that the analysis has to track. The results obtained from our workload do not yet provide enough performance motivation to explore these questions.

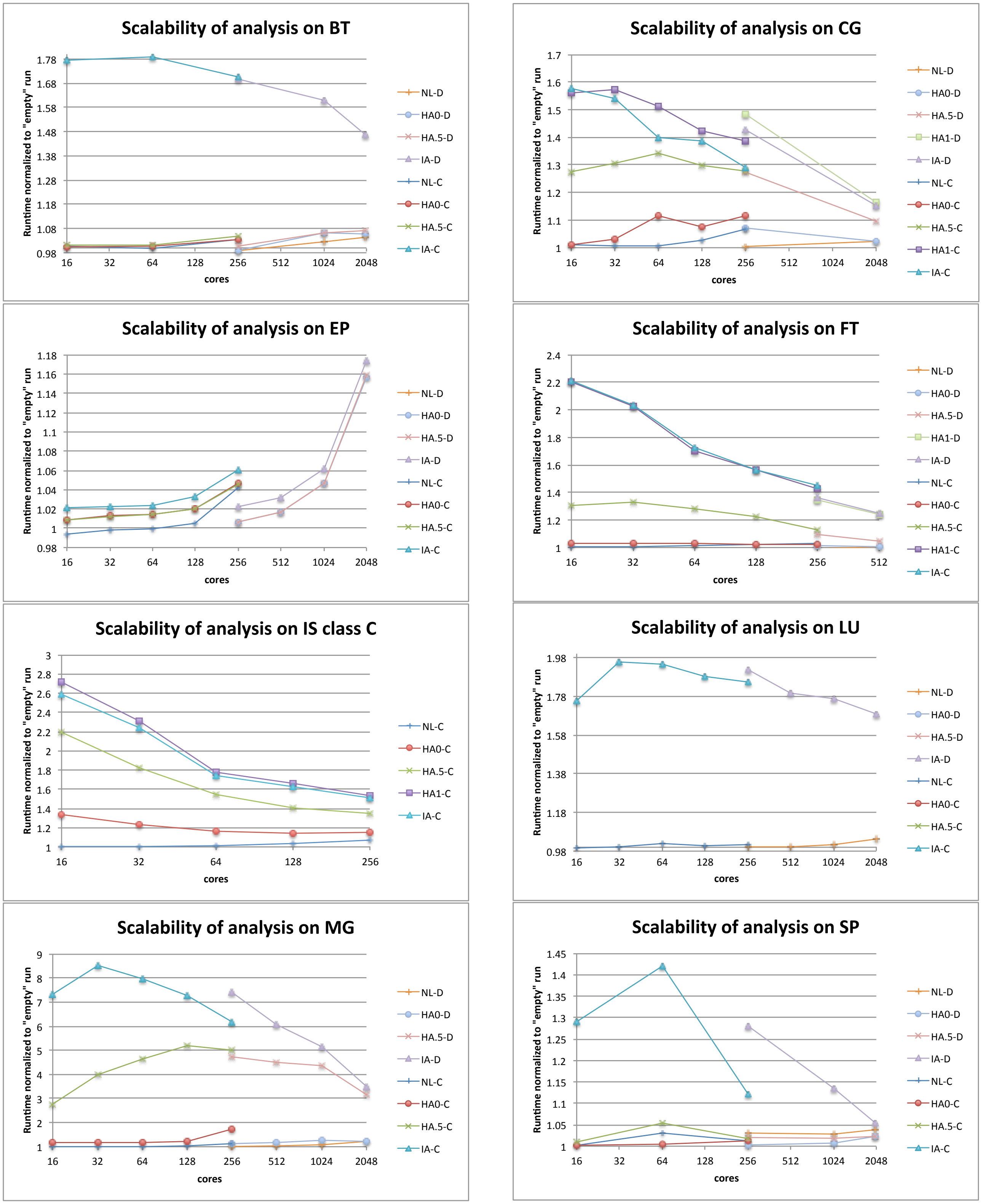
For our future work we plan to extend the data race detection implementation to provide maximum coverage on a time budget: our goal is find the maximum number of data races with no more than a guaranteed application slowdown. Our preliminary experiences indicate that we are likely to be able to guarantee no more than 2X slowdown. Toward controlling time, the scalability analysis of the internal data structures has already yielded valuable insights which allows us to derive space/state bounds. Toward improving coverage, we plan to use and perhaps augment the concept of region stacks introduced in the formalism presented in Section 4.2. We plan to experiment with several other strategies besides exponential backoff at reference and function level: i) proportional sampling per unique region stack; ii) k-region context sampling; and iii) proportional sampling at functions and exponential backoff at statements.

8. Other Related Work

Data race detection tools can be broadly classified as using static or dynamic techniques. Static techniques [9, 15, 30, 36] are scalable and complete, i.e. find all the races in the program. On the other hand, they report a very large number of false positives which need to be filtered by users and can handle only limited types of synchronization primitives such as locks or barriers.

Dynamic techniques for finding concurrency bugs can be classified into two classes: predictive techniques and precise techniques Predictive dynamic techniques [11, 39, 46] could predict concurrency bugs that did not happen in a concurrent execution; however, such techniques still report false warnings. UPC-Thrille implements a predictive dynamic technique, followed by automatic filtering of false positives. Precise dynamic techniques, such as happens-before race detection [2, 12, 14, 16, 29] and atomicity monitoring [17, 27, 47], are capable of detecting concurrency bugs that actually happen in an execution. Therefore, these techniques are precise, but they cannot give good coverage as predictive dynamic techniques.

Dynamic techniques have to address the challenge of high runtime overhead. Sampling approaches to reduce instrumentation overhead have been discussed throughout this paper. Techniques to reduce the computation overhead have been explored as well. Choi et al [11] discuss static analysis techniques to reduce the overhead of data race detection for Java programs. As alias and pointer analysis for C based programs is notoriously conservative, these techniques need to be supplemented by the runtime techniques



Scalability of analysis on BT

Scalability of analysis on CG

Scalability of analysis on EP

Scalability of analysis on FT

Scalability of analysis on IS class C

Scalability of analysis on LU

Scalability of analysis on MG

Scalability of analysis on SP

Figure 4. *Scalability of the different sampling methods when running the tool on the NAS Parallel Benchmarks, classes C and D. The overhead of instruction sampling* **I** *is high and it has been omitted for presentation purposes.*

presented in Section 4.1. Recently, Raman et al [37] describe a scalable implementation for data race detection in Habanero Java programs implemented using fine-grained structured parallelism. Their benchmarks are equivalent to our fine-grained benchmarks, while our NAS benchmarks use coarse grained interactions. They report analysis overheads as high as 10X and provide valuable data about the scalability of other state of the art race detectors for multi-threaded programs: Eraser [39] and FastTrack [16]. They report slowdowns as high as 100X for the latter.

So far there have been a lot of research effort to verify and test concurrent and parallel programs written in Java and C/pthreads for non-HPC platforms; the literature listed above supports this fact. There have also been efforts to test and verify HPC programs, mostly focused on C/MPI programs. ISP [42] is a push-button dynamic verifier capable of detecting deadlocks, resource leaks, and assertion violations in C/MPI programs. DAMPI [44, 45] overcomes ISP's scalability limitations and scales to thousands of MPI processes. Like ISP, DAMPI only tests for MPI Send/Recv interleavings, but runs in a distributed way. In contrast to our work, DAMPI instruments and reasons only about the ordering of Send/Recv operations with respect to the MPI ranks, and not about the memory accessed by these operations. Both ISP and DAMPI assume that program input is fixed. TASS [40] removes this limitation by using symbolic execution to reason about all possible inputs to a MPI program, but it is work only at inception. MPI messages can be intercepted and analyzed for bugs and anomalies. Intel MessageChecker [13] does a post-mortem analysis after collecting message traces, while MARMOT [23] and Umpire [43] check at runtime.

9. Conclusion

To our knowledge, we discuss the first implementation of a data race detector for distributed memory programs that tracks all memory references. To this end, we extend the UPC-Thrille tool to monitor, in addition to the communication operations, all memory references in a UPC program. The goal of our implementation is to provide low overhead with good program coverage when running at scale.

Dynamic program analysis tools, as implemented in UPC-Thrille, face the challenge of instrumentation overhead. UPC-Thrille implements instruction sampling, while the state-of-the-art technique to reduce overhead is considered to be function sampling. We use a workload containing UPC programs and experiment with function and instruction level sampling. Our results indicate that function level sampling is not feasible for scientific programs: increasing the input set increases the amount of work per function invocation in these applications and the analysis does not terminate. Instruction sampling works better for scientific programs and our implementation finds races with up to 65X slowdown.

We propose two techniques to improve the scalability of data race detection in UPC programs:) hierarchical function and instruction level sampling; and 2) exploiting the runtime persistence of aliasing and locality in UPC applications. The aliasing heuristic is common to PGAS languages and the combination of the two techniques is widely applicable to Chapel, Titanium, X10. Hierarchical sampling is a generic technique that is applicable orthogonal to the language, programming model or data race detection algorithm. The results indicate that both techniques are required in practice: well optimized instruction sampling introduces overheads as high as 6500% (65X slowdown), while each technique in separation is able to reduce it to 1000% (10X slowdown). When applying the optimizations in conjunction our tool finds races with at most 50% overhead when running on 2048 cores of a CrayXE6 system.

References

[1] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE computing*, December 1996.

[2] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *18th International Symposium on Computer architecture (ISCA)*, pages 234–243. ACM, 1991.

[3] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, 2001.

[4] M. Arnold, M. T. Vechev, and E. Yahav. Qvm: An efficient runtime for detecting defects in deployed systems. *ACM Trans. Softw. Eng. Methodol.*, 21(1), 2011.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[6] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. *Technical Report NAS-95-010, NASA Ames Research Center*, 1995.

[7] S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Haifa Verification Conference*, pages 208–223, 2005.

[8] D. Bonachea. Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet. Technical Report LBNL-56495, Lawrence Berkeley National Lab, October 2004.

[9] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *Programming Language Design and Implementation (PLDI)*, pages 12–21, 2007.

[10] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, and K. W. E. Brooks. Introduction to UPC and language specification, 1999.

[11] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Programming language design and implementation (PLDI)*, pages 258–269, New York, NY, USA, 2002. ACM.

[12] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst.*, 13(4):491–530, 1991.

[13] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. In *Software engineering for high performance computing system applications*, SE-HPCS '05, pages 78–82, New York, NY, USA, 2005. ACM.

[14] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Workshop on Parallel and Distributed Debugging*, 1991.

[15] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 25(2–3):199–240, 2004.

[16] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Programming language design and implementation (PLDI)*. ACM, 2009.

[17] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Programming language design and implementation (PLDI)*, pages 293–303. ACM, 2008.

[18] E. N. Hanson and T. Johnson. The interval skip list: A data structure for finding all intervals that overlap a point. In *Workshop on Algorithms and Data Structures*, pages 153–164. Springer, 1992.

[19] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th*

international conference on Architectural support for programming languages and operating systems, ASPLOS-XI, 2004.

[20] P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the berkeley upc compiler. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, 2003.

[21] P. Joshi, M. Naik, C.-S. Park, and K. Sen. An extensible active testing framework for concurrent programs. In *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer, 2009.

[22] A. Kamil and K. Yelick. Hierarchical pointer analysis for distributed programs. In *The 14th International Static Analysis Symposium (SAS 2007, Kongens Lyngby*, 2007.

[23] B. Krammer, M. Müller, and M. Resch. Runtime checking of MPI applications with MARMOT. In *Mini-Symposium Tools Support for Parallel Programming, ParCo 2005, Malaga, Spain, September 12 - 16, 2005.*, 2005.

[24] W. Kuchera and C. Wallace. The UPC memory model: Problems and prospects. In *the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2004.

[25] B. Liblit and A. Aiken. Type systems for distributed data structures. In *In the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL*, pages 199–213, 2000.

[26] B. Liblit, A. Aiken, and K. Yelick. Type systems for distributed data sharing. In *In International Static Analysis Symposium*. SpringerVerlag, 2001.

[27] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. *SIGARCH Comput. Archit. News*, 34(5):37–48, 2006.

[28] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *PLDI*, 2009.

[29] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing*, pages 24–33. ACM, 1991.

[30] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[31] The NAS Parallel Benchmarks. Available at http://www.nas.nasa.gov/Software/NPB.

[32] The UPC NAS Parallel Benchmarks. Available at http://upc.gwu.edu/download.html.

[33] R. H. B. Netzer and B. P. Miller. What are race conditions? some issues and formalizations. *LOPLAS*, 1992.

[34] S. Olivier and J. Prins. Scalable dynamic load balancing using UPC. In *International Conference on Parallel Processing (ICPP)*, ICPP '08, 2008.

[35] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu. Efficient Data Race Detection for Distributed Memory Parallel Programs. In *Proceedings of the Supercomputing Conference (SC11)*, 2011.

[36] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *Programming language design and implementation (PLDI)*, pages 14–24. ACM, 2004.

[37] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, PLDI '12, 2012.

[38] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, 2006.

[39] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[40] S. F. Siegel and T. K. Zirkel. Automatic formal verification of MPI-based parallel programs. In *Principles and practice of parallel programming*, PPoPP '11, pages 309–310, New York, NY, USA, 2011. ACM.

[41] C. Terboven. Comparing intel thread checker and sun thread analyzer. In *PARCO'07*, pages 669–676, 2007.

[42] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP: a tool for model checking MPI programs. In *Principles and practice of parallel programming*, PPoPP '08, pages 285–286, New York, NY, USA, 2008. ACM.

[43] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[44] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *Supercomputing*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[45] A. Vo, G. Gopalakrishnan, R. M. Kirby, B. R. de Supinski, M. Schulz, and G. Bronevetsky. Large scale verification of mpi programs using lamport clocks with lazy update. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 330–339, Washington, DC, USA, 2011. IEEE Computer Society.

[46] C. von Praun and T. R. Gross. Object race detection. In *Object oriented programming, systems, languages, and applications (OOPSLA)*, pages 70–82. ACM, 2001.

[47] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Not.*, 40(6):1–14, 2005.

[48] K. Yelick, D. Bonachea, and C. Wallace. A Proposal for a UPC Memory Consistency Model. Technical Report LBNL-54983, Lawrence Berkeley National Laboratory, May 2004.